CineLog: Уеб приложение за проследяване на филми

Автор: **Есат Мустафа**
Факултетен номер: [redacted]
Използвани AI технологии: *Claude Sonnet 4.5* и *Gemini 2.0 Flash Experimental*

Въведение и идея

Идеята за този проект, да си призная, дойде от лична нужда. Аз често гледам филми, но ми липсваше начин систематично да проследявам какво съм гледал, какво ми е харесало и какви мисли съм имал за тях. Затова реших да направя **CineLog**. Целта беше да създам **уеб инструмент**, който да съчетава автоматично извличане на информация за филми с възможност за персонални записи и оценки.

Основната ми цел не беше **GenAI** да напише проекта. Исках да покажа, че един разработчик може да ползва `GenAI` като силен помощник, като същевременно *мисли критично* и сам взема решения за архитектурата. По време на разработката винаги давах конкретни задачи, като след това преглеждах резултатите. Ще отбележа, че `GenAI` често излишно усложняваше логиката на кода. Затова съм описал точно къде съм променил кода или къде съм заменил AI решението с мое.

CineLog работи просто. Потребителят може да добави филми, като напише заглавието, след това приложението автоматично тегли информация като жанр, режисьор и постер посредством `OMDb API`. След това може да се зададе лична оценка от 1 до 10, да се проследява статуса и да се оставят бележки. Управлението става с търсене, филтриране и редактиране. Цялата информация се записва в `Supabase`.

Потребителите са тези, които обичат кино или тези, които искат просто по-персонална система за проследяване на какво са гледали или какво искат да гледат. Това може да са случайни зрители или студенти, които пишат по-подробни бележки. Съчетавайки автоматичната работа и личните настройки, се пести време.

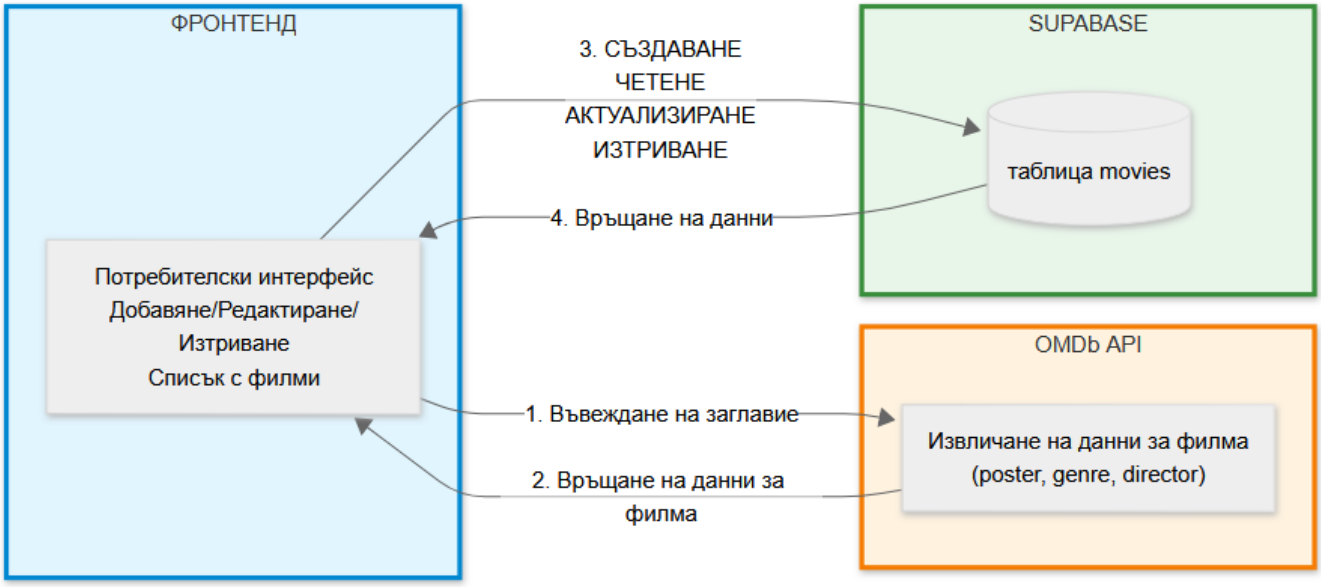
Успях да намеря няколко конкурента - **Letterboxd**, **IMDb** и **Trakt**. **Letterboxd** е супер, но е по-социален. **IMDb** позволява списъци, но липсват лични оценки и детайлни бележки. И в края **Trakt** също се крепи на социални функции. И много от тези решения искат абонамент. **CineLog** запълва този пропуск - предлага **лична**, **безплатна** и **независима платформа**.

Този проект е добър пример как **GenAI** влиза в разработката. Той беше полезен за идеи с `Tailwind CSS`, за базата данни, също така, всички технически диаграми (`Mermaid код`), които илюстрират архитектурата, бяха първоначално генерирани от AI модел. Но **човешката преценка беше задължителна**. Налагаше се преработка и вземане на архитектурни решения. И именно това беше **ключът към качеството**.

Функционалности

CineLog е създаден като **лично приложение** без никакви социални функции. Приложението предлага ясен и фокусиран инструмент за колекцията, като предоставя висока степен на потребителски контрол върху данните. В центъра на цялата функционалност стои **CRUD моделът**. Това значи четири ключови действия: добавяне на филми, разглеждане на колекцията, редактиране на информация и изтриване на филми.

Процесът на добавяне е двустепенен. При добавяне на филм, потребителят въвежда заглавие (и може да добави и година). След това се натиска бутонът „Fetch from OMDb“. Това кара приложението да се свърже с **`API`**, което автоматично попълва информация като жанр, режисьор, постер и IMDb рейтинг. След това потребителят добавя лични данни: оценка от 1 до 10, статус на гледане и бележки. Едва тогава се запазва записът. Този сложен поток на данни, който показва как **`React App`** комуникира с **`OMDb`** и **`Supabase`**, е визуализиран в **„Диаграма на последователността“**.

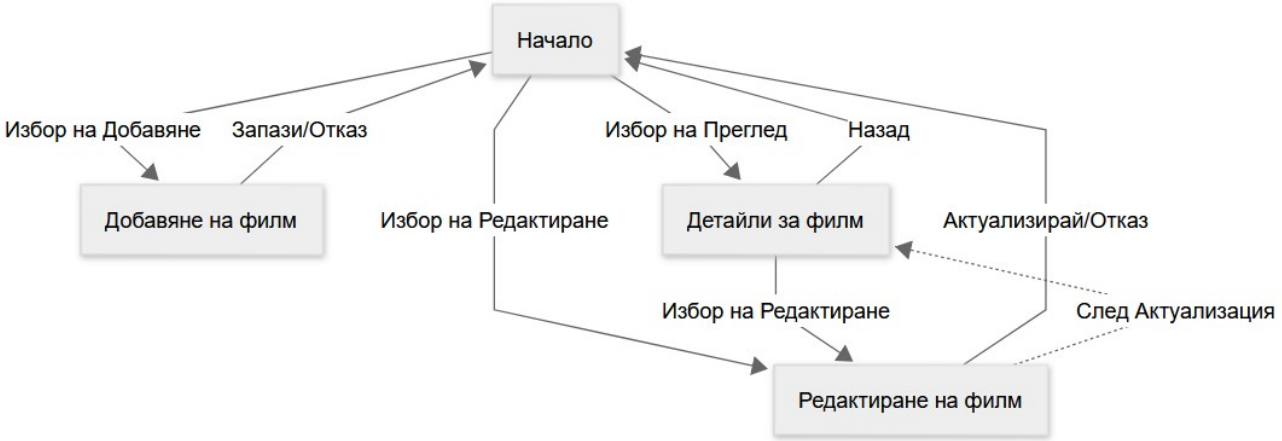


[Диаграма на последователността - Процес „Добавяне на нов филм“]

Източник: Генерирано от *Claude Sonnet 4.5*, модифицирано и адаптирано от автора.

Разглеждането на колекцията става на началната страница (**`Home`**), където всички запазени филми са показани в адаптивна решетка. Тук се ползва търсене по заглавие. Може да се филтрира по статус (**`To Watch`**, **`Watching`**, **`Watched`**) и да се сортира по заглавие, година или по лична оценка. Интерфейсът дава резултати **в реално време**.

Редактирането на филм позволява промяна само на оценка, статус и бележки. Оригиналната информация от **`OMDb`** остава **само за четене**. Изтриването става само след потвърждение. Кликването върху бутона **View** отваря детайлен изглед (**`Movie Details`**). Преходите между тези страници и възможните състояния са описани в **„Диаграма на навигацията“**.



[Диаграма на навигацията]

Източник: Генерирано от *Claude Sonnet 4.5*, модифицирано и адаптирано от автора.

---

movies		
uuid	id	Първичен ключ, автоматично генериран UUID
text	title	Заглавие на филма (NOT NULL)
integer	year	Година на издаване
text	genre	Жанрове на филма
text	director	Име на режисьор
text	poster_url	URL адрес на изображение на постер
text	imdb_rating	IMDb рейтинг
smallint	personal_rating	Потребителска оценка 1-10
varchar	status	Статус: To Watch, Watching, Watched
text	notes	Лични бележки

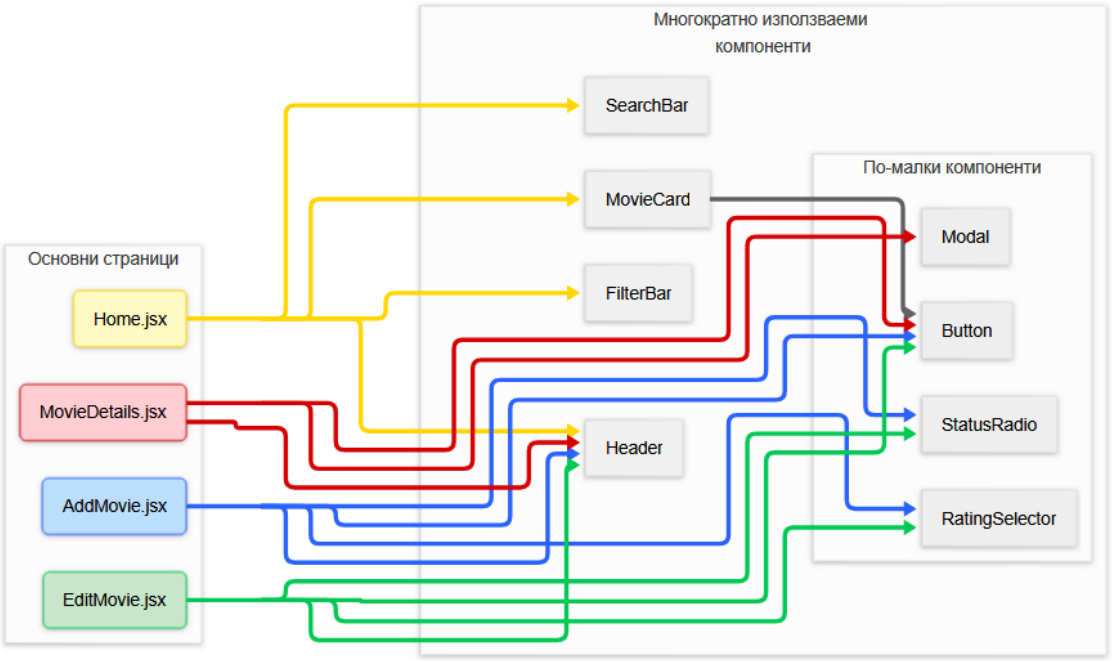
В основата на всичко стои базата данни – **Supabase**. Тя съхранява всички филми. Основната таблица има две групи данни. Първата група са данни от **`OMDb`** (**`title`**, **`year`**, **`genre`**, **`director`**, **`poster_url`**, **`imdb_rating`**). Втората група са личните данни, които се въвеждат: **`personal_rating`**, **`status`**, **`notes`**. Разбира се, имаме и **`id`** (уникален идентификатор). Цялостната схема на таблицата е представена в **„Схемата на базата данни“**.

[Схема на базата данни]

Източник: Генерирано от *Claude Sonnet 4.5*, модифицирано и адаптирано от автора.

Приложението е структурирано йерархично чрез **`React`** компоненти. Имаме основните страници (**`Home`**, **`Add Movie`**, **`Edit Movie`**, **`Movie Details`**). След това имаме многократно използваеми компоненти (**`MovieCard`**, **`SearchBar`**, **`FilterBar`**, **`Header`**). Имаме и по-малки компоненти за формите (бутони, модални прозорци, селектор за оценка). Този модулен подход помага много за поддръжката и е мястото, където **`GenAI`** помага за генерирането на **`Tailwind CSS`** класовете. Връзките между тези компоненти са детайлно показани в **„Диаграма на**

компонентната архитектура“.



[Диаграма на компонентната архитектура]

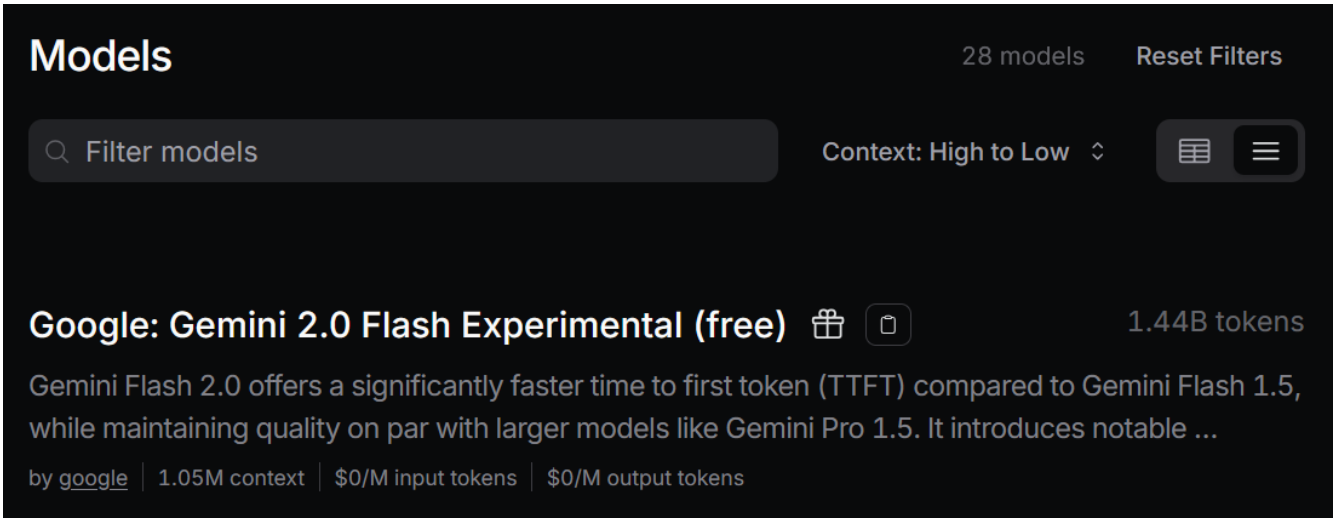
Източник: Генерирано от *Claude Sonnet 4.5*, модифицирано и адаптирано от автора.

CineLog се отличава от **Letterboxd**, **IMDb** и **Trakt** по няколко причини. Той е **прост**, има **чист интерфейс** и се фокусира само върху **личната колекция**, няма социални разсейвания. Потребителят получава автоматично попълнени данни, но контролира личните си бележки и оценки, търсенето и филтрирането са **в реално време**. Основното предимство е, че е напълно **свободен**, **независим** от платформа и е ориентиран към **личната употреба**. **CineLog** е идеален за хора, които искат ясен и организиран начин да следят филмите си.

Разработка

Преди да започна реалната работа по **CineLog**, трябваше внимателно да подбера *инструментите*, които ще използвам през целия цикъл на разработка. Исках технология, която да ми даде **бързина** и **ефективност**, особено предвид факта, че щях да разчитам много на AI помощ за генериране на стилове и оптимизиране на компонентната структура.

Още в самото начало обърнах специално внимание на избора на подходящ AI модел. Тъй като планирах да използвам AI за **`Tailwind CSS`** класове и структурни предложения в **`React`**, беше важно моделът да може да **чете цели файлове**, да разбира **логически връзки** и, разбира се, да бъде **безплатен**, тъй като проектът е студентски. В **`OpenRouter`** филтрирах наличните модели по „Context: High to Low“. Най-отгоре в списъка се появи **`Google: Gemini 2.0 Flash Experimental (free)`**.



Този модел беше **идеален**: голям контекст, бързи отговори (`Flash`), стабилно разбиране на код и нулева цена. Интегрирах го в работния си процес чрез `Cline` в `VSCode`, което ми позволи да подавам кратки инструкции като *„Style this with Tailwind CSS“*, без да обяснявам целия компонент.

След като приключих с избора на AI асистент, преминах към останалите инструменти. За фронтенда избрах `React` с `JSX`, за да изградя *интерактивен интерфейс*. За стилове - `Tailwind CSS`, комбиниран с AI помощ за бързи *utility* класове. За базата данни - `Supabase`, който осигурява `PostgreSQL` с удобен `JavaScript` клиент. Интегрирах `OMDb API` за филмови метаданни, `Lucide React` за леки `SVG` икони и `React Router` за навигацията.

След като реших технологичния стек, стартирах проекта с `Vite`, понеже е значително по-бърз от `Create React App`:

```
npm create vite@latest cinelog -- --template react
```

В този момент имах само началната `React` страница, работеща на `http://localhost:5173`.

След това преминах към `Tailwind CSS`. Първо опитах автоматична инсталация чрез `Gemini 2.0 Flash`, но това доведе до проблеми с `PostCSS`:

```
[plugin:vite:css] [postcss] It looks like you're trying to use `tailwindcss`
directly as a PostCSS plugin…
```

След проверка в официалната документация реших да инсталирам `Tailwind` **ръчно** чрез официалния `Vite plugin`. Следвайки стъпките от ръководството, `Tailwind` беше добавен успешно.

За да подредя проекта още от самото начало, организирах структурата в `/src` по възможно най-ясен начин:

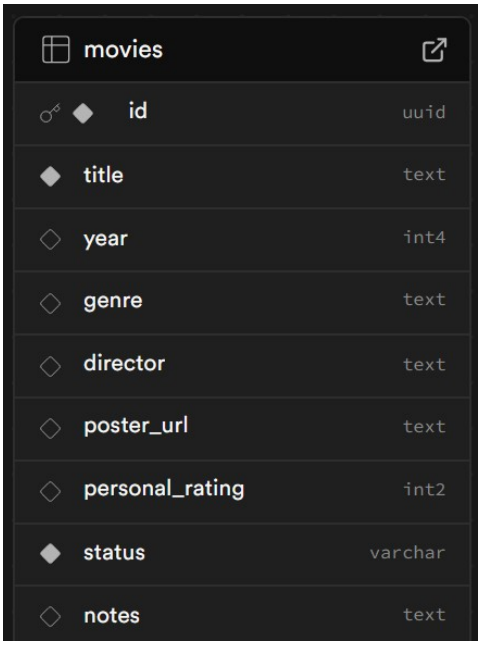
- `pages/` - основни страници

- `components/` - многократно използваеми `React` компоненти
- `lib/` - `API` логика, помощни функции и конфигурации

Навигацията между страниците я реализирах чрез `React Router DOM`. Инсталацията беше стандартна:

```
npm install react-router-dom
```

Огърнах кореновия компонент с `BrowserRouter`, а в отделен файл `AppRoutes.jsx` дефинирах маршрутизацията: `Home (/)`, `AddMovie (/add)`, `EditMovie (/edit/:id)` и `MovieDetails (/movie/:id)`. Всички страници се визуализираха правилно, което **потвърди**, че конфигурацията е успешна.



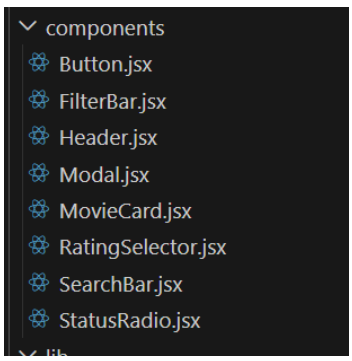
След оформянето на основната структура преминах към базата данни - **Supabase**. Идеята беше проста: **CineLog** се нуждае от лека, но надеждна база за съхранение на филми, рейтинги и бележки, без *сложни backend операции*. `Supabase` пасна идеално.

Създадох нов проект в `https://supabase.com`, избрах име „CineLog“ и изчаках базата да се инициализира. Когато проектът беше готов, от `Table Editor` създадох таблица `movies`, базирана на предварително планираната структура.

За да свържа `React` приложението със `Supabase`, първо взех нужните *креденшъли* от Project Settings → `API`: `Project URL` и `anon public key`.

И двата са **задължителни** за клиентска връзка. След това инсталирах `Supabase` клиента:

```
npm install @supabase/supabase-js
```



Създадох нов файл `supabaseClient.js` в `src/lib`, който държи конфигурацията и експортира инстанцията. Това **улесни** достъпа до базата във всички компоненти.

Преди да премина към изграждането на началната страница, реших, че е хубаво да подготвя **базовата архитектура** за бъдещите компоненти. Във `VS Code` създадох ръчно новата директория `components` в рамките на `src`. Тази структура беше предварително дефинирана от мен (вж. **„Диаграма на компонентната**

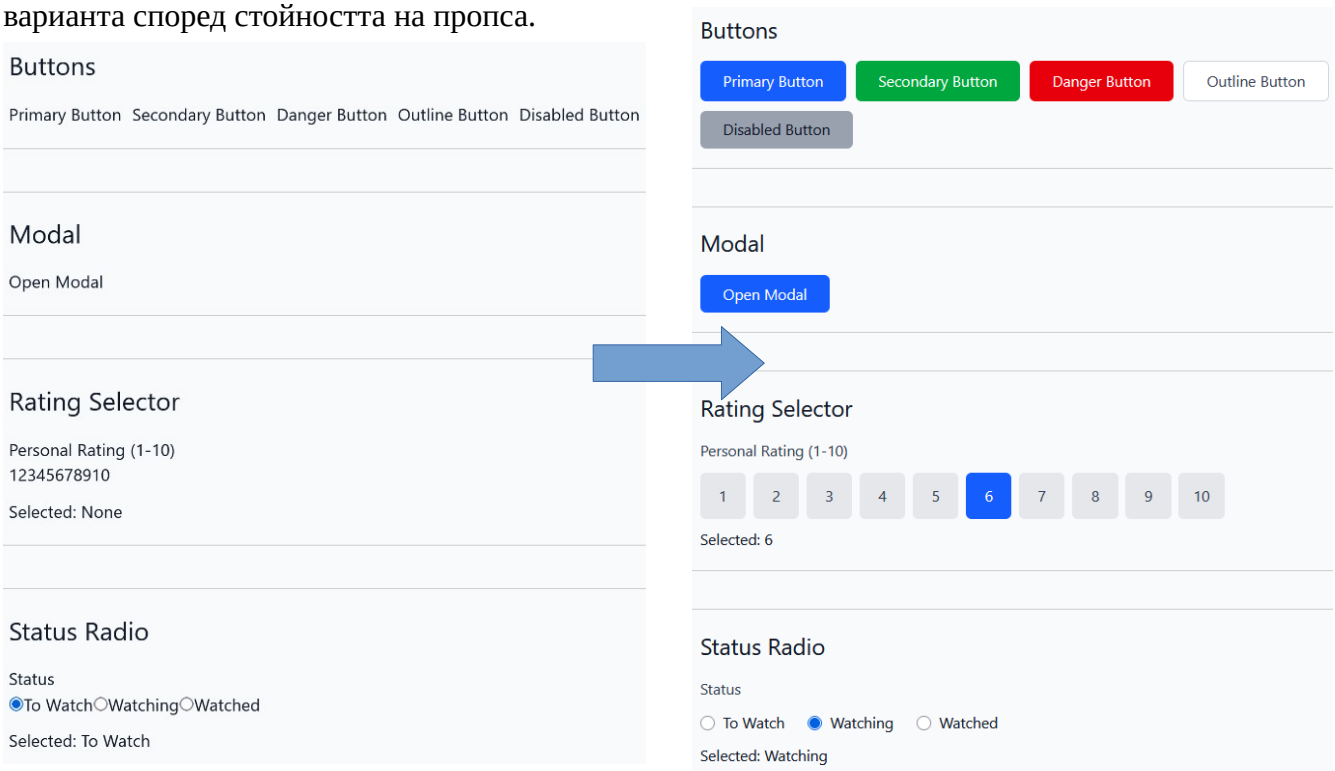
архитектура“). Създадох всички необходими файлове **ръчно**, като целта беше да установя предвидима и устойчива структура за многократно използваеми елементи.

След това започнах изграждането на *малките* компоненти. Те трябваше да бъдат използвани повторно. Наложи се да дефинирам базовата логическа структура за **`Button`**, **`Modal`**, **`RatingSelector`** и **`StatusRadio`**. Работих без стилове, като целта беше само да установя свойствата и вътрешната функционалност. **`Button`** трябваше да поддържа всички варианти на поведение в приложението чрез *пропсове*: за съдържание, тип вариант, събития при натискане, състояние **`disabled`** и **`HTML type`**. **`Modal`** изискваше *пропсове* за отворено състояние, функции за потвърждение и отказ, както и текстови параметри. **`RatingSelector`** предоставяше бутоните за рейтинг от 1 до 10. **`StatusRadio`** съдържаше трите статусни стойности като радио избор.

След като структурите бяха готови, отворих **`Cline`** с **`Gemini 2.0 Flash`** и предоставих достъп до директорията **`components`**. Използвах следния промпт за цялостно оформление с **`Tailwind`**:

```
“Style all components in the components folder with Tailwind. Button needs 4
color variants (primary blue, secondary green, danger red, outline white with
border), padding, rounded corners, and disabled state. Modal needs a dark
overlay background with a white centered card, title, message text, and two
buttons at the bottom. RatingSelector needs a label and 10 number buttons in a
row, with different colors for selected vs unselected. StatusRadio needs a
label and 3 radio options in a row. Make everything look consistent with a
blue theme.”
```

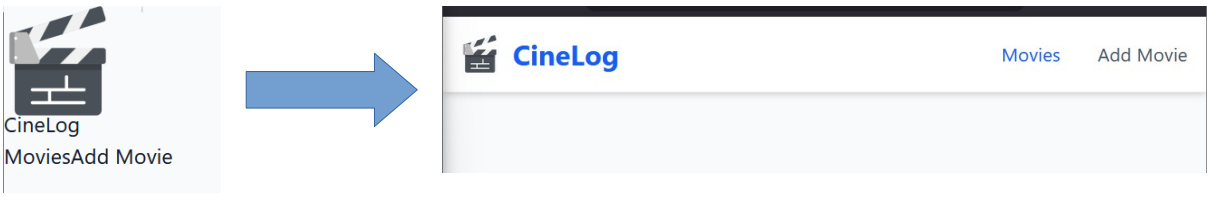
Gemini генерира оформени версии за четирите компонента, които коректно имплементираха логиката на *object lookup* за вариантите на бутона. **Modal** беше оформен като *центриран диалог* с полупрозрачен фон, а **RatingSelector** и **StatusRadio** визуализираха ясно разграничение между избрани и неизбрани стойности. След като прегледах детайлно кода, открих, че логиката за **disabled** състоянието в **Button** е прекалено усложнена. Извърших една промяна. Използвах тернарен оператор, който избира между класове за **disabled** или класове за варианта според стойността на пропса.



След това започнах изграждането на основните компоненти, които трябваше да използват вече създадените елементи. Първо създадох **Header**, който щеше да функционира като навигационна лента, достъпна на всички страници. Използвах **useLocation**, за да получа текущия път, след което дефинирах помощна функция **isActive** за сравнение на пътищата. Дадох на **Gemini** следния промпт:

```
“Style this Header component with Tailwind. Make it stick to the top of the page. Add a white background with a bottom border. Put the logo and text on the left side, navigation links on the right. Make the current page link blue, others gray. Add hover effects on the links.”
```

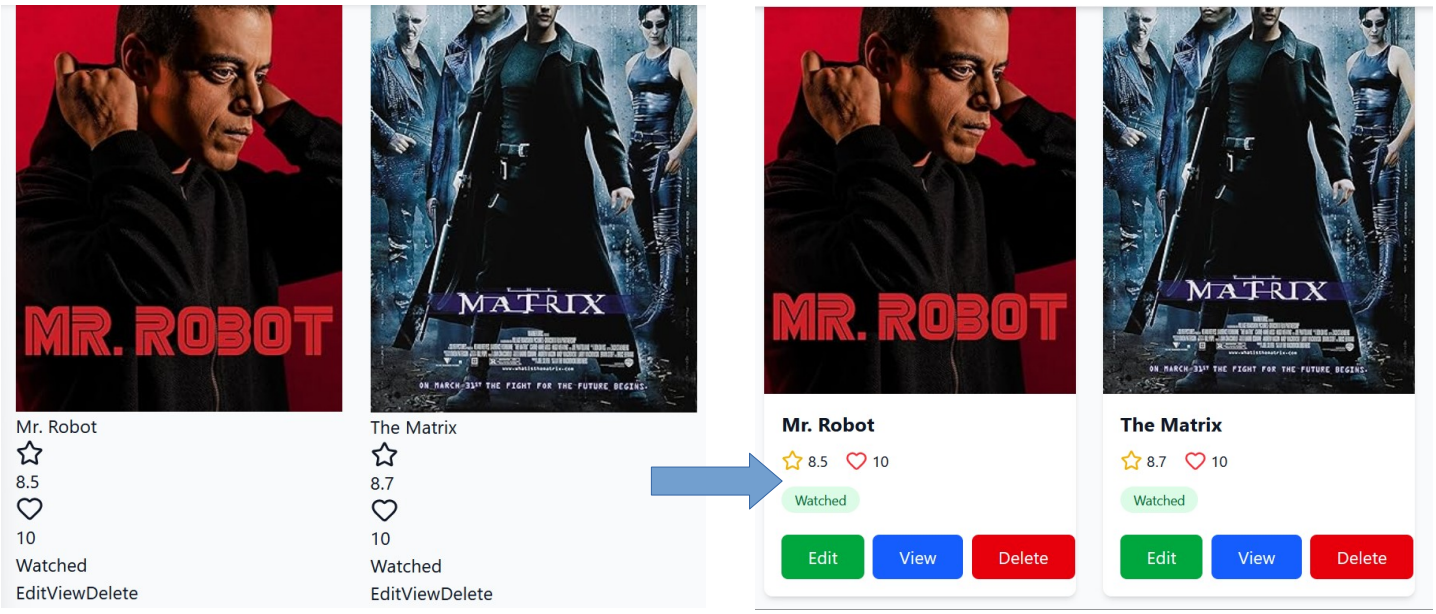
Gemini генерира оформен компонент със *sticky позициониране*, долна граница и сенки, както и ясно структурирана навигация. Извърших една промяна. Изведох повтарящата се логика за класове на навигационните линкове в отделна помощна функция **linkClasses**.



Продължих с **`MovieCard`**. Компонентът трябваше да визуализира филм в мрежата на началната страница. Използвах **`Button`** и иконите **`Star`** и **`Heart`** от **`Lucide React`**. Добавих **`useNavigate`** за управление на пренасочванията и подадох следния промпт:

```
“Style this MovieCard with Tailwind. Make it a card with rounded corners and shadow. Poster image at the top, movie info below. Show title, ratings with icons, status badge with colors, and three buttons (Edit, View and Delete) at the bottom. Add a hover effect that makes the shadow bigger.”
```

`Gemini` генерира подредена карта със сенки и коректно позиционирани икони. Направих две редакции: Първо преместих логиката за стиловете на статусите в помощна функция **`getStatusClass`**, реализирана чрез *object lookup*. И второ коригирах синтаксиса на шаблонните литерали при навигационните извиквания и замених разпределението чрез **`margin`** с *flex gap*.

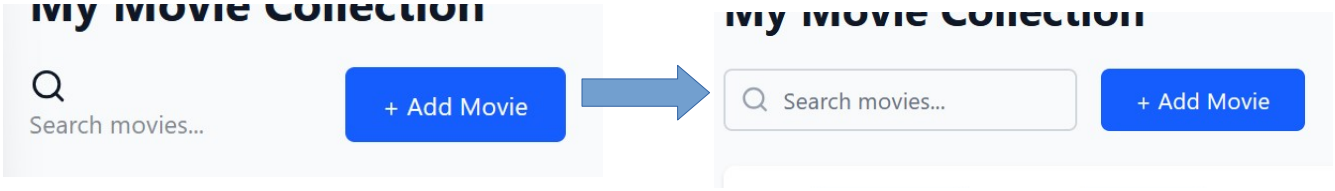


Продължих с реализацията на **`SearchBar.jsx`**. Компонентът трябваше да работи като *контролиран вход* с икона. Използвах иконата **`Search`** от **`Lucide React`** и дефинирах *пропсовете* **`searchQuery`** и **`setSearchQuery`** и подадох следния промпт:

```
“Style this SearchBar with Tailwind. Put a search icon on the left inside the input field. Make the input take full width with padding, border, and rounded corners. When focused, change the border to blue. Make it look clean and modern.”
```

`Gemini` произведе пълно съответствие с изискванията. Позиционирането на иконата

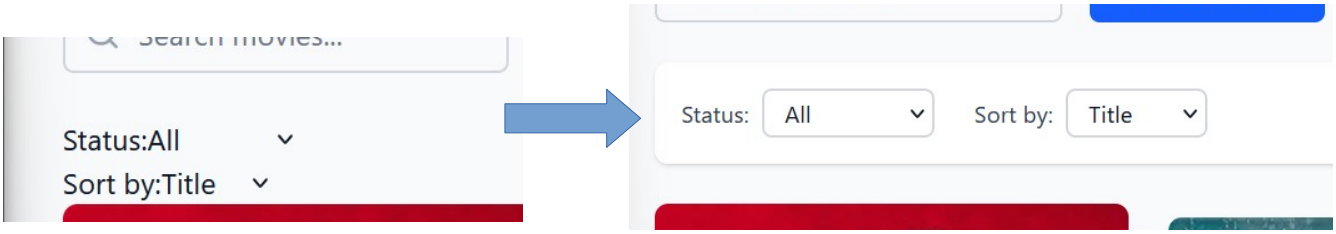
работеше коректно чрез **absolute** и **transform**. Входът използваше достатъчно вътрешни отстояния, за да компенсира иконата. Поведението при фокус следваше моделите от останалите компоненти.



След това създадох **FilterBar**. Компонентът трябваше да обработва два контрола: филтриране и сортиране. Подадох следния промпт:

```
“Style this FilterBar with Tailwind. Create a horizontal layout with two
dropdown filters (Status and Sort). Add labels next to each dropdown. Put
everything in a white card with padding and shadow. Make the dropdowns have
borders and change border color when focused.”
```

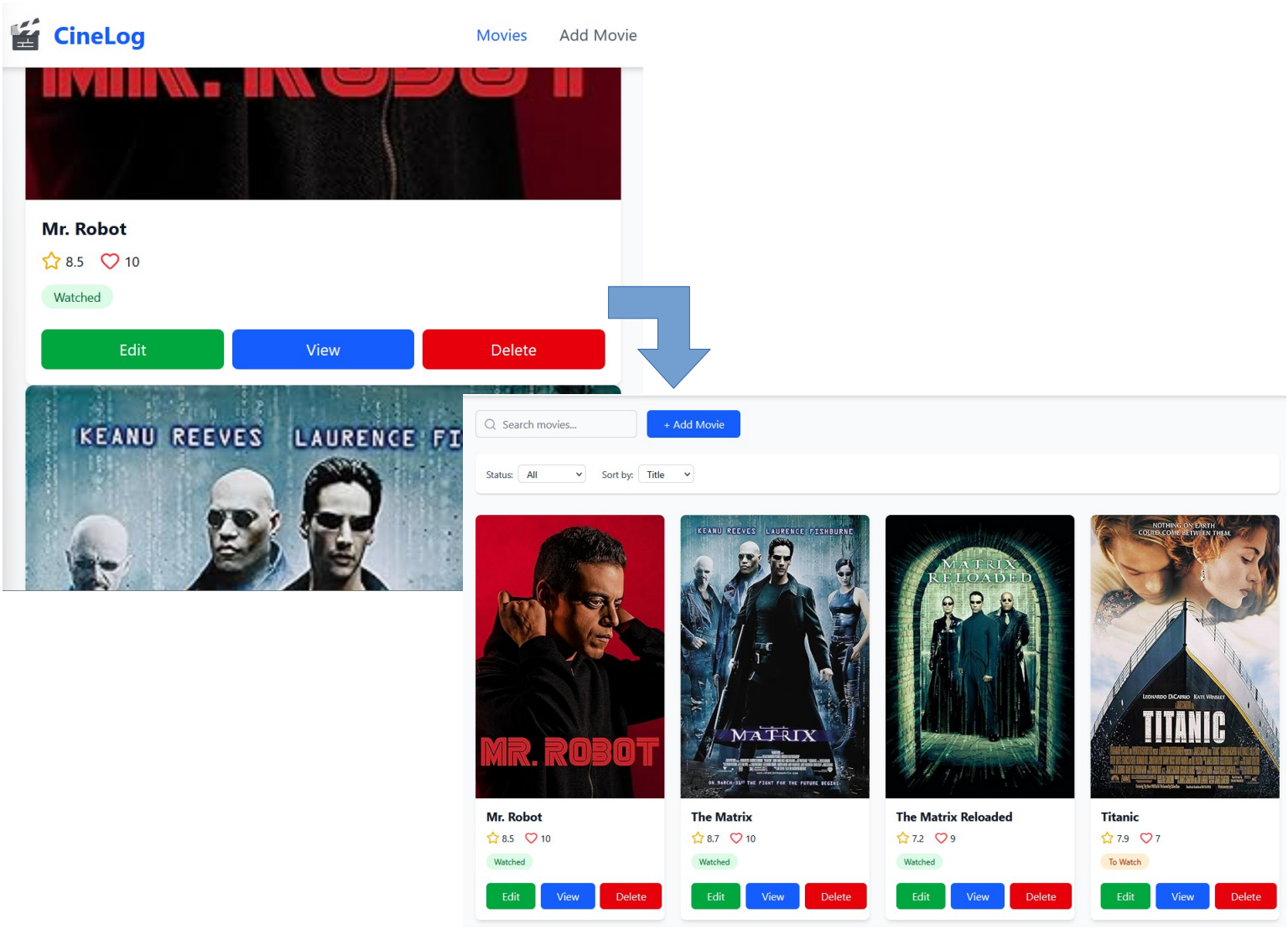
Генерираният вариант осигури **последователно оформление** с бяла карта, фина сянка и адаптивен *flex layout*. Dropdown елементите бяха равномерно подравнени и работеха като *контролирани полета*.



Създадох **Home.jsx** като основна страница. Импортирах **Header**, **SearchBar**, **FilterBar**, **MovieCard** и **Button**. Управлявах множество състояния: списък от филми, филтрирани резултати, **searchQuery**, статусен филтър, сортиране, **loading** и **error**. Подадох следния промпт:

```
“Style this Home page with Tailwind. Add a page title at the top. Put the
search bar and 'Add Movie' button in a row. Below that, show movies in a grid
- 1 column on mobile, 2-3 on tablet, 4 on desktop. Add messages for when there
are no movies or when loading. Make it responsive.”
```

Gemini генерира подредена и *отзивчива* структура. Направих няколко редакции. Премахнах излишни *wrapper елементи* около **SearchBar**. Добавих explicit **variant="primary"** към всички основни бутони. Внедрих **Retry бутон** в **error** състоянието. Актуализирах визуализацията на грешките с червен текст за по-ясна сигнализация.

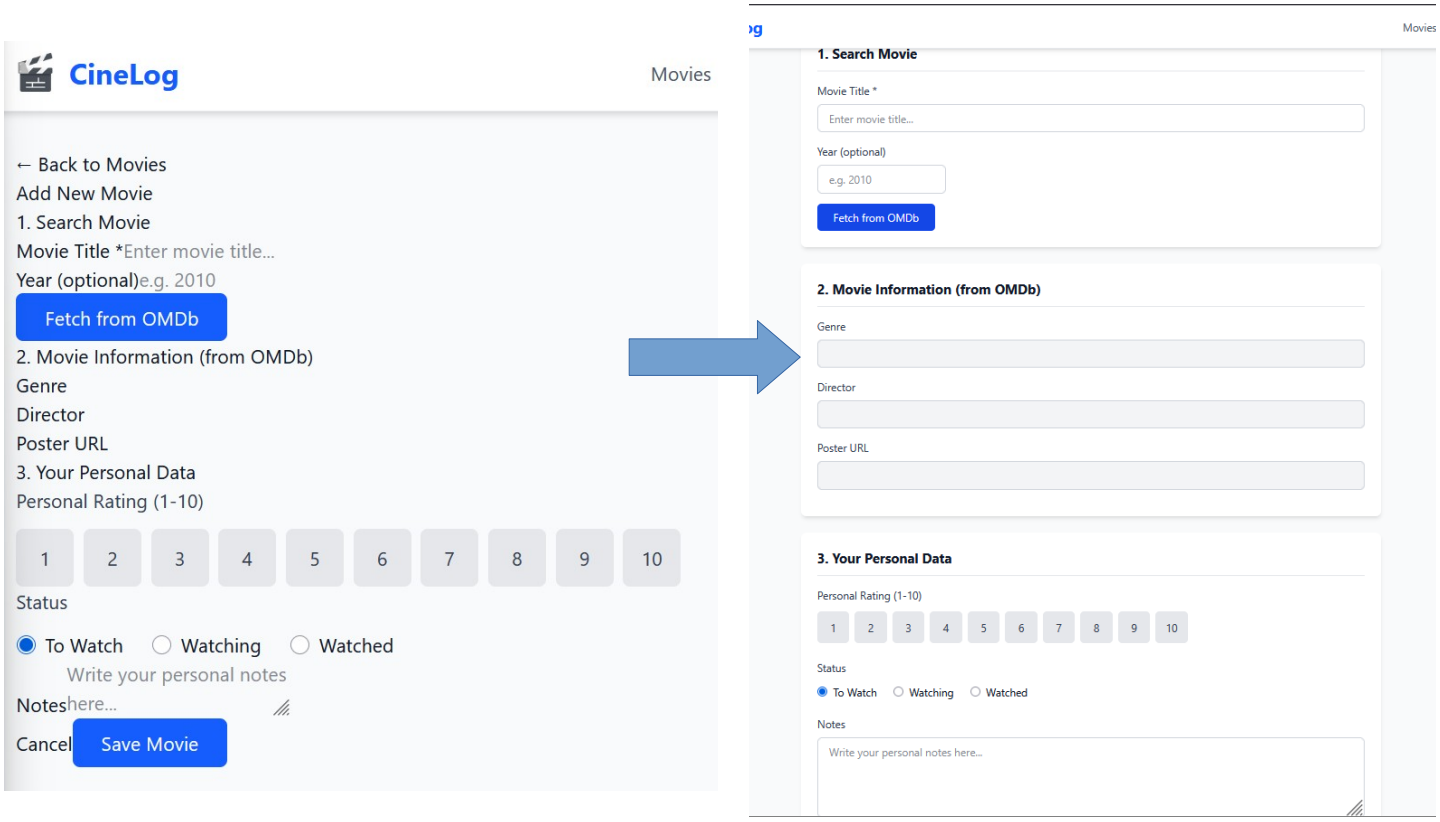


Продължих с **`AddMovie.jsx`**. Страницата беше разделена на три секции: търсене в **`OMDb`**, визуализиране на върнатата информация и попълване на персонални данни. Използвах **`RatingSelector`**, **`StatusRadio`** и **`Button`**. Добавих логика за валидиране и поведение на формата при промяна на заглавието. Подадох следния промпт:

```
“Style this AddMovie page with Tailwind. Divide it into 3 sections with white
cards. Section 1: movie search with title and year inputs plus a fetch button.
Section 2: show fetched data (genre, director, poster) in disabled/gray
inputs. Section 3: personal data with rating buttons, status options, and
notes textarea. Put Cancel and Save buttons at the bottom.”
```

`Gemini` изгради коректна трисекционна структура със стабилно визуално отделяне. Редактирах **`Cancel`** бутона. Промених го на **`Link`** компонент, но запазих визуалното оформление чрез присвоени класове. Добавих explicit **`variant="primary"`** при бутона **`Fetch`**. Разширих

`spacing` около `RatingSelector` и `StatusRadio`.

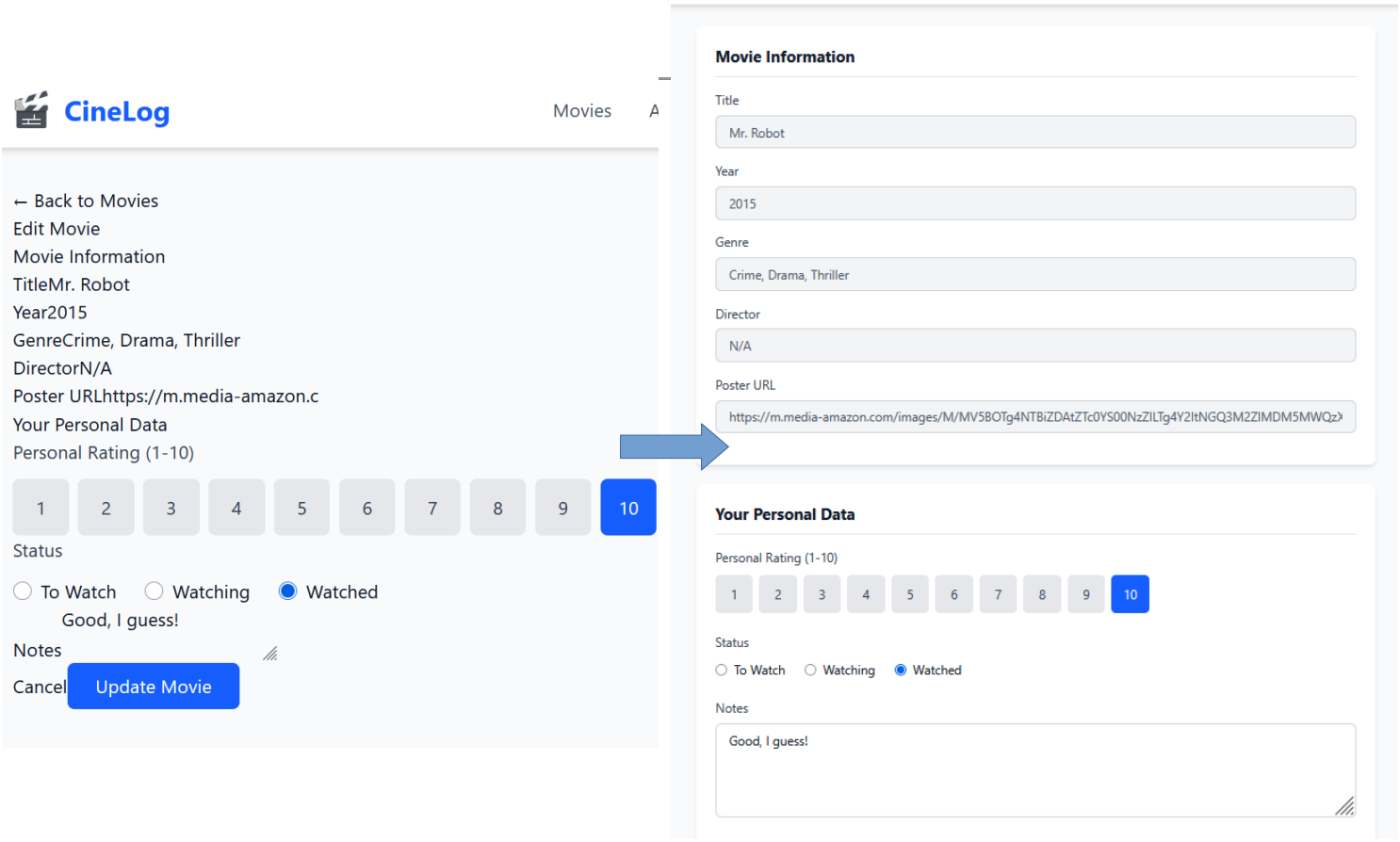


Създадох `EditMovie.jsx`. Страницата съдържа две секции. Първата предоставя неизменяеми полета с `OMDb` данни. Втората съдържа персоналните настройки. Използвах `useParams` за получаване на `ID` и извличане на данните при зареждане. Подадох следния промпт:

“Style this EditMovie page with Tailwind. Similar to AddMovie but with 2 sections instead of 3. First section shows movie info in disabled/gray fields. Second section has editable personal data (rating, status, notes). Add Cancel and Update buttons at the bottom. Show a loading message while data is being fetched.”

```
“Style this EditMovie page with Tailwind. Similar to AddMovie but with 2 sections instead of 3. First section shows movie info in disabled/gray fields. Second section has editable personal data (rating, status, notes). Add Cancel and Update buttons at the bottom. Show a loading message while data is being fetched.”
```

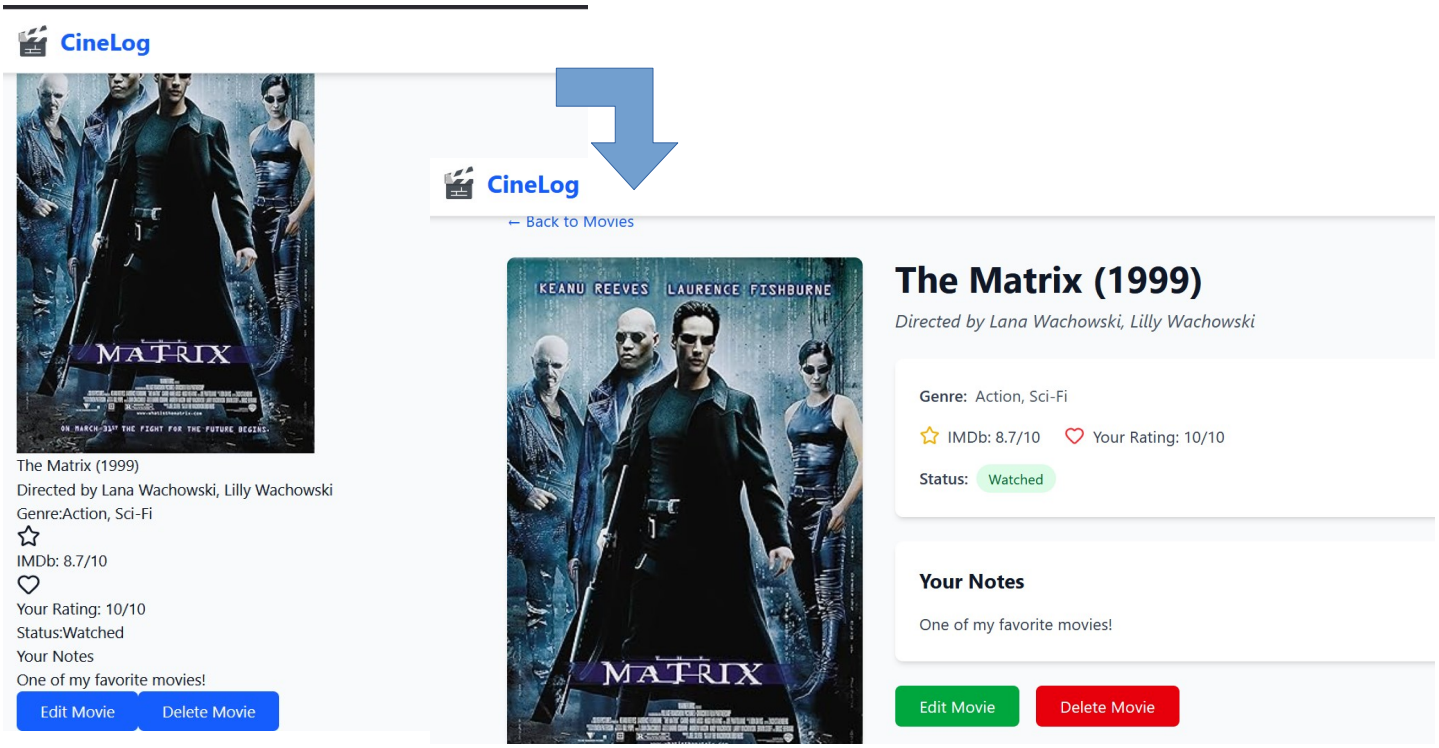
Gemini произведе оформление, което **следваше същия модел** като **AddMovie**. **OMDb** полетата бяха правилно деактивирани. Компонентът обработваше напълно редактирането и изпращането на актуализирани данни.



Завърших секцията с **MovieDetails.jsx**. Компонентът извличаше детайлна информация за един филм и визуализираше постер, метаданни и потребителски данни. Включих **Star** и **Heart** иконите, **Button** и **Modal**. Дефинирах **getStatusClass**, за да се запази логиката от **MovieCard**. Подадох следния промпт:

```
“Style this MovieDetails page with Tailwind. Put a large poster on the left side, movie details on the right. Show title, director, genre, ratings with icons, status badge, and personal notes in separate white cards. Add Edit and Delete buttons below. Make it stack vertically on mobile screens.”
```

Gemini произведе **пълно оформен компонент** с двуколонна структура, която се адаптира за мобилни устройства. Иконите използваха предвидените размери и цветове. Картичките за информация и бележки следваха оформлението.



След като приключих с дефинирането на всички компоненти, дойде време за **интеграцията на данните** и имплементирането на пълния **CRUD** функционал. Първоначално създадох няколко помощни модула в директорията `src/lib/`, които поеха цялата логика за базата данни и външния `API` - `omdbApi.js` и `movieApi.js`. В omdbApi.js дефинирах функцията `fetchMovieByTitle` за заявки към `OMDb`, която връща форматирани филмови метаданни. В `movieApi.js` изведох основните **CRUD** операции към `Supabase`: `getAllMovies()`, `getMovieById(id)`, `createMovie(movieData)`, `updateMovie(id, movieData)` и `deleteMovie(id)`.

`Home.jsx` пое управлението на състоянието, импортирайки всички тези функции. При зареждане на страницата се извличат всички записи. Клиентското филтриране и сортиране се изпълнява чрез `useEffect`, като се обработват търсене по текст, статус и предпочитание за подредба. Страницата визуализира подходящи състояния за празни данни, грешки или продължително зареждане.

Интеграцията в `AddMovie` комбинира двата външни източника. Изтеглената информация от `OMDb` се обединява с въведените персонални стойности преди подаването ѝ към `Supabase`. Критично решение беше **да се нулират** всички `OMDb` полета при промяна на `title` или `year`, за да се избегнат неточни записи.

`EditMovie` зарежда съществуващ запис, като позволява редакция само на персоналните атрибути (`personal_rating`, `status`, `notes`). Данните, извлечени от `OMDb`, останаха **само за**

четене.

`MovieDetails` показва пълния набор данни. За потребителско потвърждение преди изтриване беше използван `Modal` компонент.

След свързването на всички страници и успешното имплементиране на всички помощни модули, приложението придоби **пълна функционалност**: завършен **CRUD** с постоянно съхранение в облак, външна `API` интеграция и търсене/филтриране **в реално време**.

След това преминах към **систематично тестване и отстраняване на грешки**. Работех в комбинация от AI анализ и ръчно тестване. Първо, за да подобря четливостта на кода и да улесня бъдещата поддръжка, **добавих коментари** в целия проект. Използвах `Gemini 2.0 Flash` в `Cline` с промпта*:*

```
“Add simple comments to component files explaining what each file does and
what the main functions do. Keep comments short and helpful.”
```

Резултатът беше полезен. Компонентите и помощните файлове получиха кратки пояснения, които **подпомагат разчитането на логиката**.

Извърших и **цялостно AI-асистирано дебъгване**. Предоставих достъп до всички `React` компоненти и помощни файлове със следния промпт:

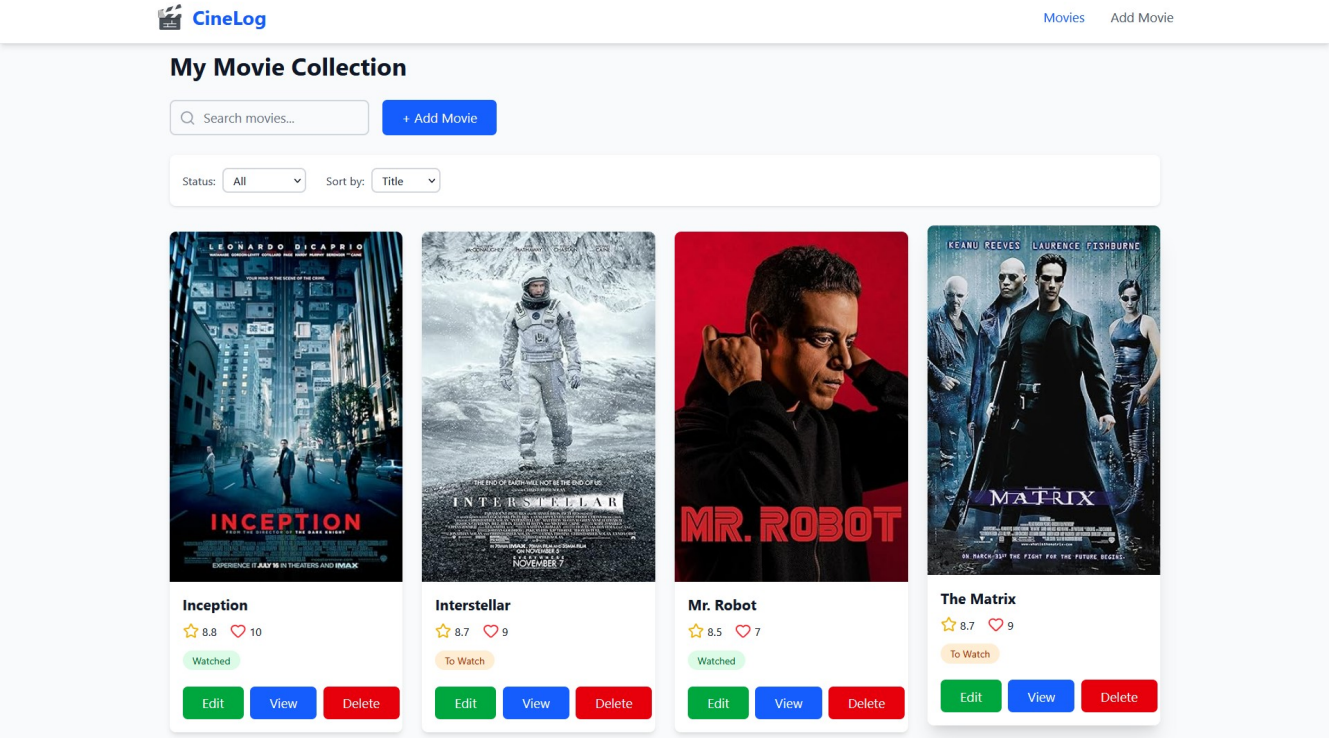
```
“Analyze all React component and page files for bugs, errors, or problems.
Look for missing error handling, logic issues, or things that might break.
Tell me what's wrong and how to fix it.”
```

Анализът откри няколко **критични несъответствия**, включително липсващи проверки за `movie.status`, изисквания за `line-clamp-2 Tailwind` плъгин, липса на валидиране на `API` ключа и прекомерна употреба на `alert()`. Анализът също така препоръча подобряване на **достъпността** в `Modal` чрез клавиатурна навигация и `ARIA` обозначения за `RatingSelector`. Всички тези проблеми бяха **коригирани незабавно** чрез добавяне на проверки за `null`, валидиране на променливите и мигриране на потвържденията за изтриване към `Modal` компонента.

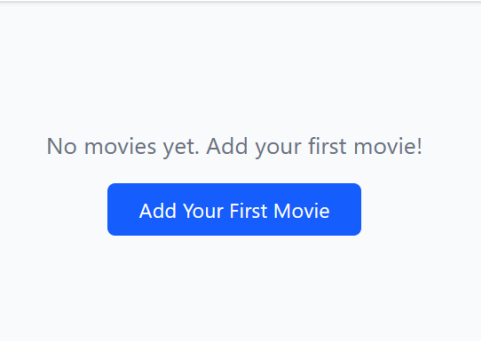
Накрая, извърших **ръчно тестване**, за да покрия всички функционални сценарии, гранични случаи и *UX поведение*. Тестването включваше добавяне, редактиране и изтриване, търсене и филтриране, както и сценарии за грешки при мрежови проблеми или липсващи входове. Приложението демонстрира **стабилна работа** във всички случаи и реагира по подходящ начин на различните гранични ситуации.

Резултати

Начална страница



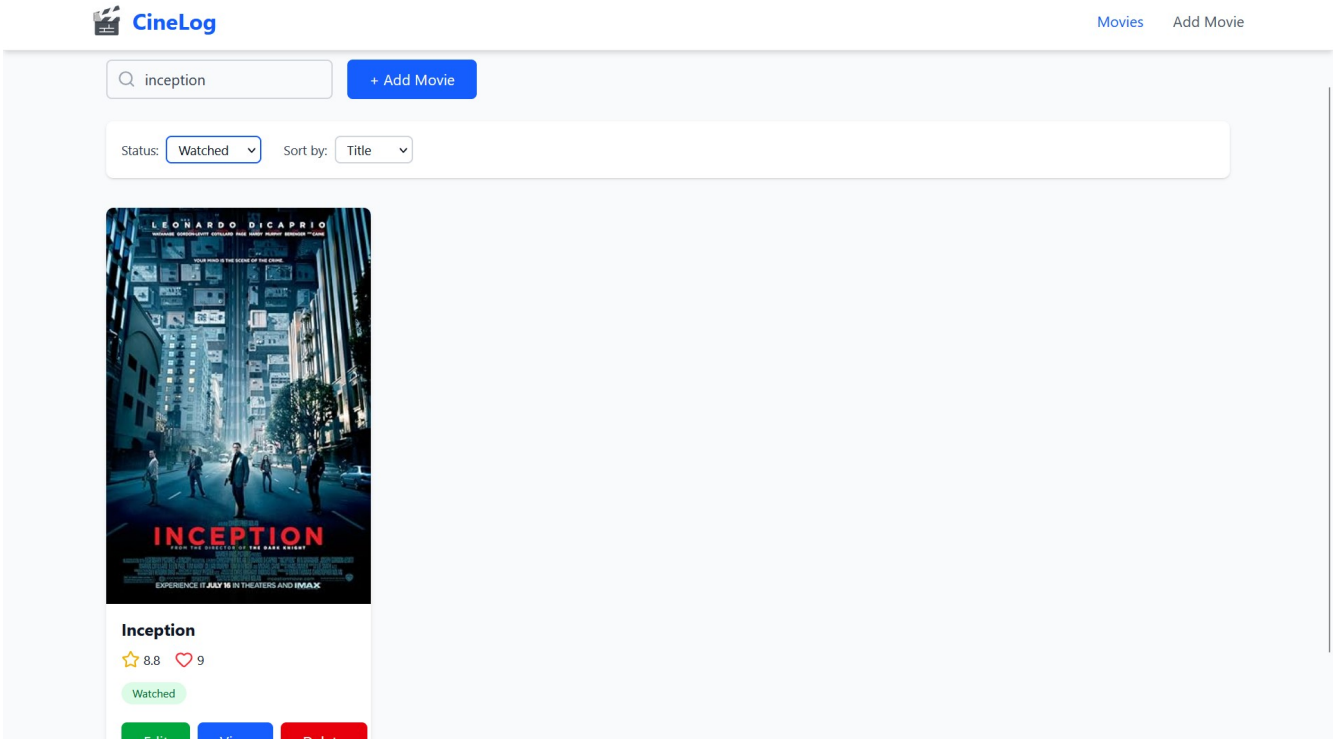
В началния изглед се визуализира **цялата колекция**. Горната фиксирана лента съдържа логото **CineLog** вляво и навигационните елементи вдясно. Под нея се вижда заглавието *My Movie Collection*. Търсачката позволява **филтриране в реално време**. Добавих бутон **`Add Movie`** непосредствено до нея. Включих и два контролни селектора - **`Status`** и **`Sort by`**. Те се комбинират със заявките от търсачката и **актуализират резултатите без презареждане**. Реших структурата на решетката да е *адаптивна*: една колона при мобилни устройства и до четири при по-широки екрани. Всяка карта представя постер, заглавие, IMDb рейтинг със звезда, персонален рейтинг със сърце и **цветова индикация** за статуса. В долната част на картата се намират трите бутона **`Edit`**, **`View`**, **`Delete`**.



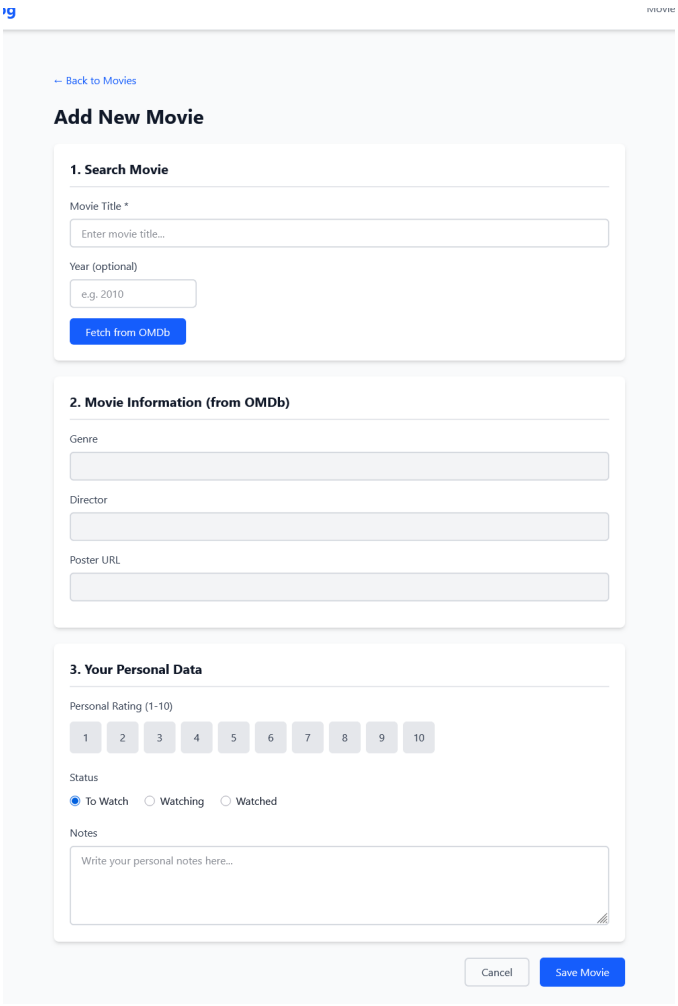
Празна колекция

При липса на данни потребителят вижда **празно състояние**. Съобщението е ясно. Придружаващо действие също е налично. То насочва към добавяне на първи запис. Районирах елемента в центъра за ясна визуализация.

Търсене и филтриране в действие

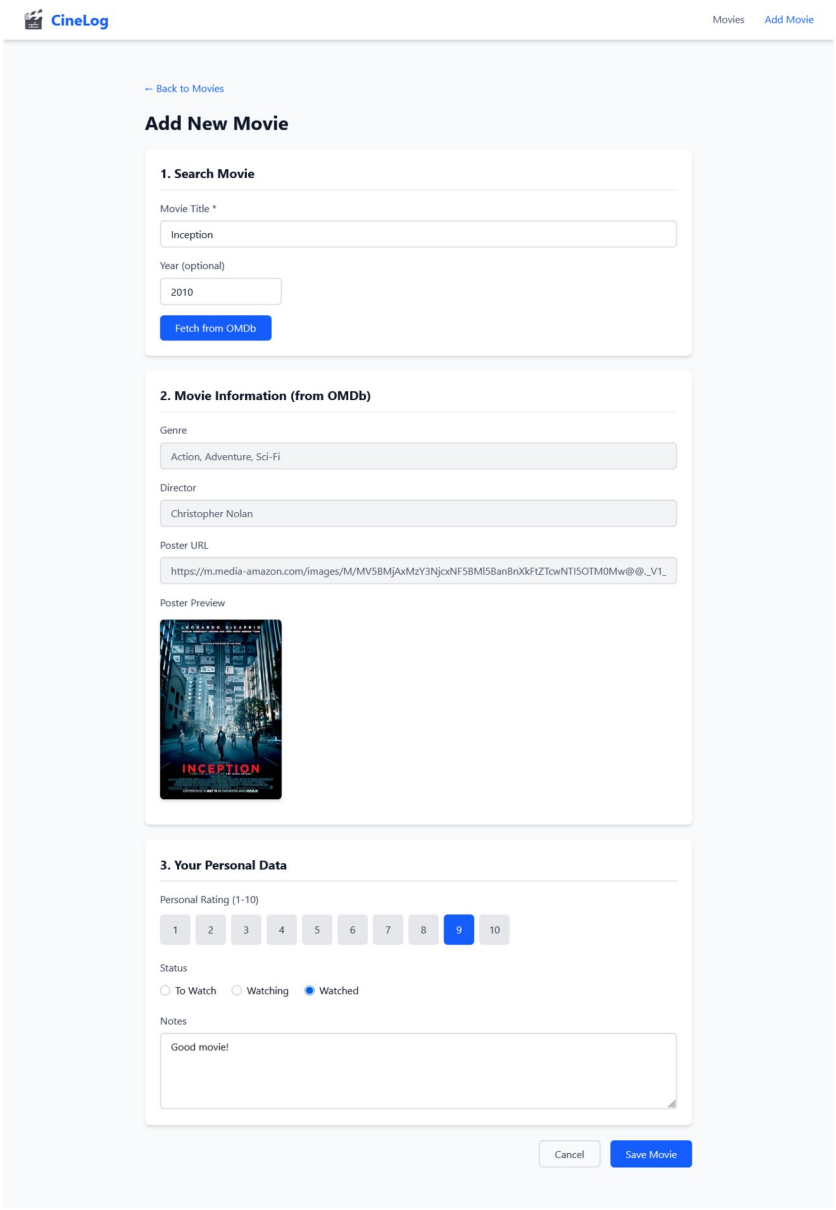


Търсенето и филтрите действат **динамично**. Въвеждането на *inception* веднага ограничава резултатите до близките заглавия. Комбинацията със статус филтъра стеснява набора допълнително. При липса на съвпадения се извежда съобщение *No movies match your filters*. Това поведение беше **важно за постигане на прецизност**.



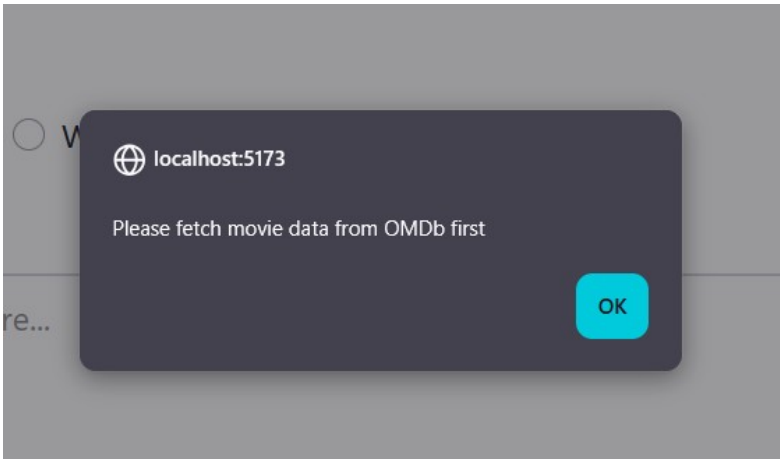
Добавяне на филм

Страницата за добавяне е разделена на **три области**. В първата се въвеждат `Title` и `Year`. Бутонът `Fetch from OMDb` остава активен при наличие на въведено заглавие. Втората област показва празни, деактивирани полета за `Genre`, `Director` и `Poster URL`. Третата включва персоналните данни - рейтинг селектор с десет стойности, радио избор за статус и поле за бележки.



След извличане от OMDb

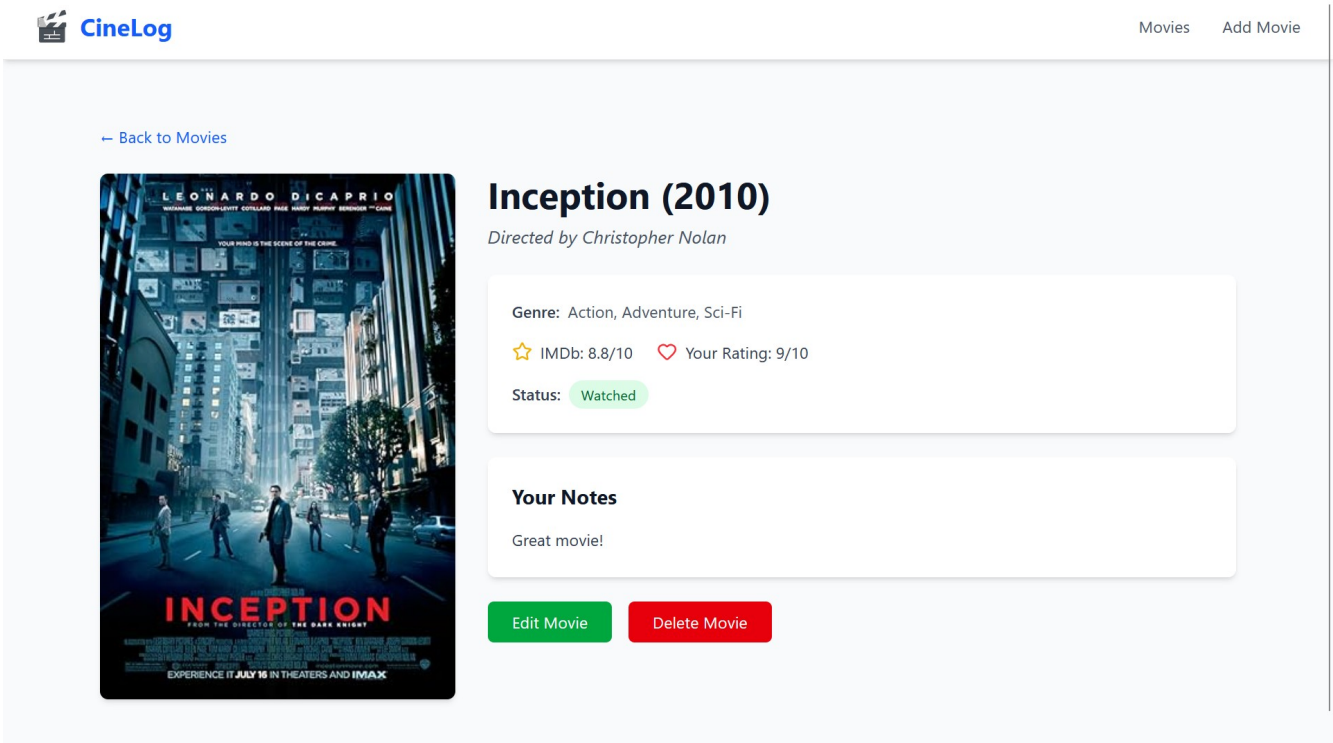
След извличане чрез **OMDb** втората секция се попълва автоматично. Появява се и **преглед на постера**. Това значително съкращава ръчния труд. Секцията за персонални данни остава **интерактивна**. Рейтингът се маркира при избор. Статусът може да се променя. Бележките се въвеждат свободно. При запис се проверява наличието на извлечени данни. И накрая записът се изпраща към **Supabase** и се връща към основния изглед.



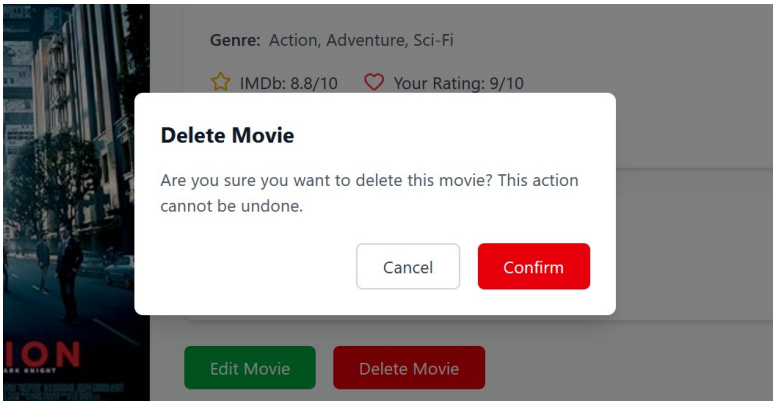
Грешка при валидация

При опит за запис без предварително извличане се визуализира **предупреждение**. Същото се случва и при несъществуващо заглавие в `OMDb`.

Страница с детайли за филм



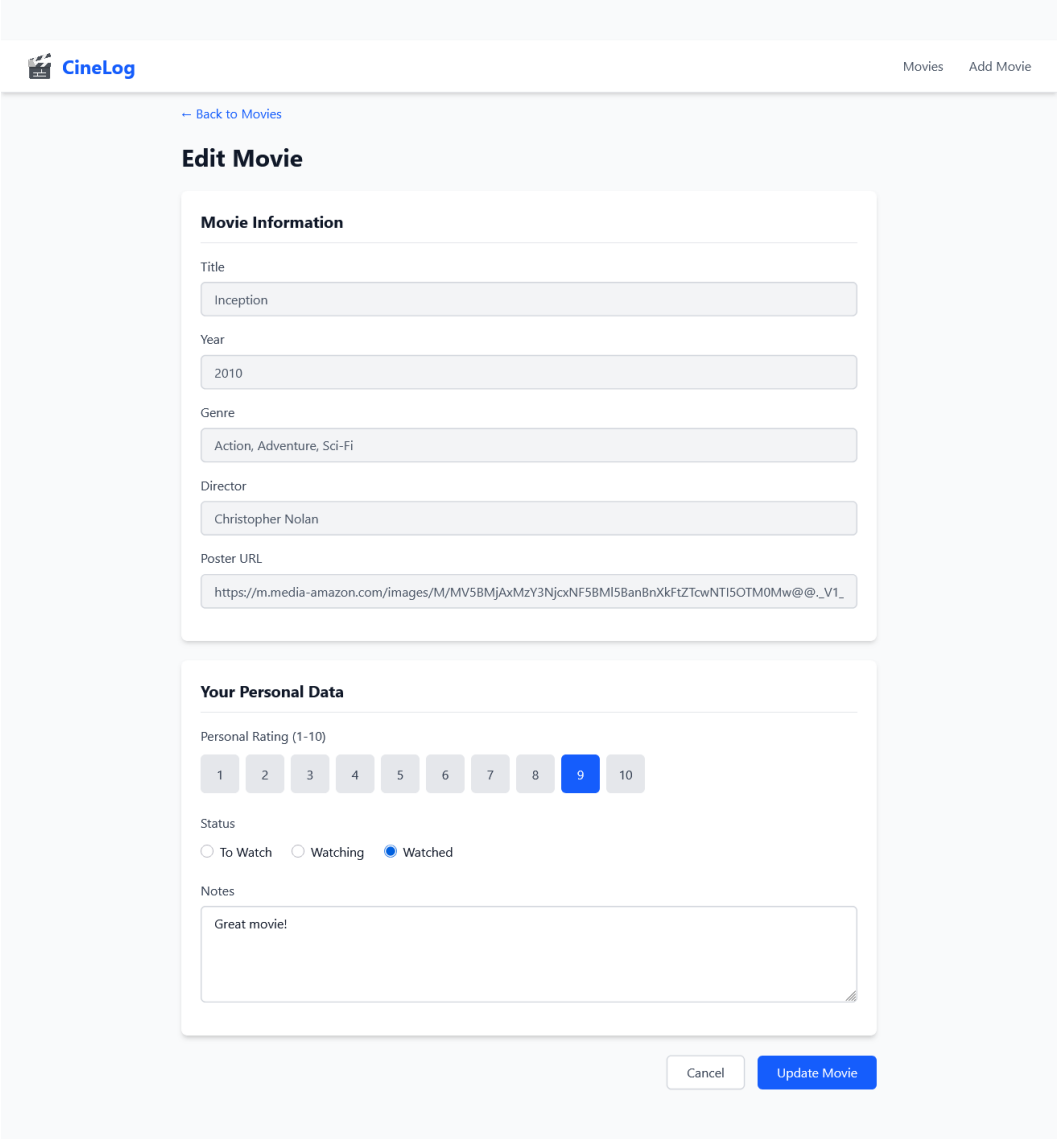
На страницата за детайлен преглед използвах **двуколонна структура**. Отляво се позиционира голям постер. Отдясно се подреждат всички информационни елементи – заглавие, година, режисьор, информационна карта с Genre, IMDb рейтинг, персонален рейтинг и статус, в отделна карта се показват бележките. Ако липсват, се визуализира празно състояние. В дъното са бутоните `Edit Movie` и `Delete Movie`.



Модален прозорец за потвърждение на изтриване

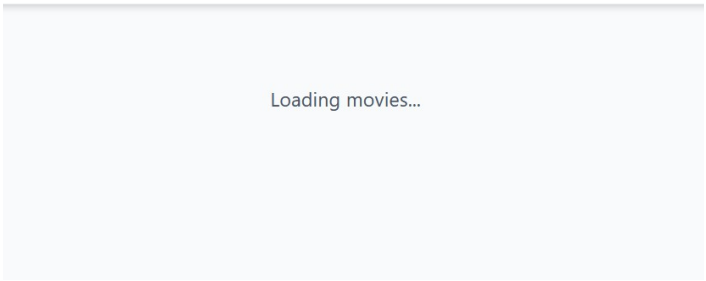
Модалното потвърждение се активира при изтриване. Модалният прозорец е оформен като **централно разположена карта** върху полупрозрачен фон. Като `Cancel` прекъсва процеса и `Confirm` извършва окончателно премахване и връща към списъка.

Форма за редактиране на филм



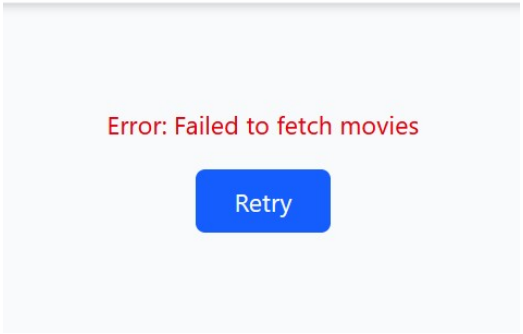
Страницата за редакция показва всички `OMDb` данни в **неактивни полета**. Целта е да

предотвратя промяна на източниковите данни. Редактират се само персонални стойности. При `Update Movie` се актуализират единствено `rating`, `status` и `notes`. Действието навигационно връща към основната страница.



Състояние на зареждане

При зареждане се визуализира **текстов индикатор**. Формите също показват *Fetching* или *Saving* при операции.



Състояние на грешка с опция за повторен опит

При грешки системата изписва съобщение и показва бутона **Retry**. Това намалява необходимостта от презареждане на страницата и осигурява по-добро **управление на откази**.

Заключение

Разработката демонстрира **методично приложение на GenAI** като подпомагащ инструмент в жизнения цикъл на софтуерното създаване. Аз не използвах AI за автоматично генериране на приложение. Вместо това интегрирах неговите възможности в критични моменти, като запазих **пълен контрол** върху архитектурните решения и качеството на кода. Първо формулирах идеи след това **валидирах подходи с** AI, и едва тогава пристъпвах към финално имплементиране.

Проектът премина през ясни етапи. Дефинирах самостоятелно схемата и структурата. Включих `AI` за ускоряване на повтаряеми задачи (`Tailwind` стилизация, добавяне на коментари и т.н). Всички взаимодействия бяха *прецизни и целенасочени*. Винаги подлагах получените резултати на **критична оценка**. Често откривах повтарящи се шаблони или усложнена логика, което налагаше **ръчен рефакторинг** и **оптимизация**.

Техническите решения показват самостоятелна преценка. Технологичният стек беше избран от мен. Конструирах потребителските потоци и компонентната архитектура. Интеграцията на **два външни източника** – `OMDb` и `Supabase` – надхвърля обичайните **CRUD** сценарии.

В началото работех върху малки компоненти. След това започнах да използвам по-глобални *промптове*, които оптимизираха работния процес. Фазата на дебъгване беше особено показателна. Разчитах на **аналитичните способности на AI**, но запазих контрол върху преценката и корекциите.

Тази работа изпълнява целта да използвам **GenAI осмислено и критично**. **AI** ускори процеси, които не изискваха концептуално мислене. Същинските **архитектурни решения** останаха зависими от моя анализ. Резултатът е демонстрация как **AI** може да бъде интегриран като *методологичен елемент* в разработката, при условие че се прилага с **критична оценка** и ясно разбиране на проектните цели.